Boiboite Opener Framework

Release 1.0.0

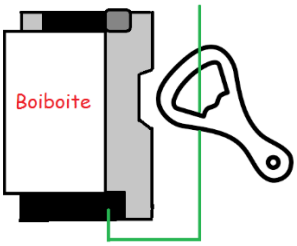
Sep 02, 2022

User manual

CHAPTER 1

Introduction

1.1 Overview



BOF (Boiboite Opener Framework) is a testing framework for industrial and field protocols implementations and devices. It is a Python 3.6+ library that provides means to send, receive, create, parse and manipulate frames from supported protocols, for basic interaction as well as for offensive testing.

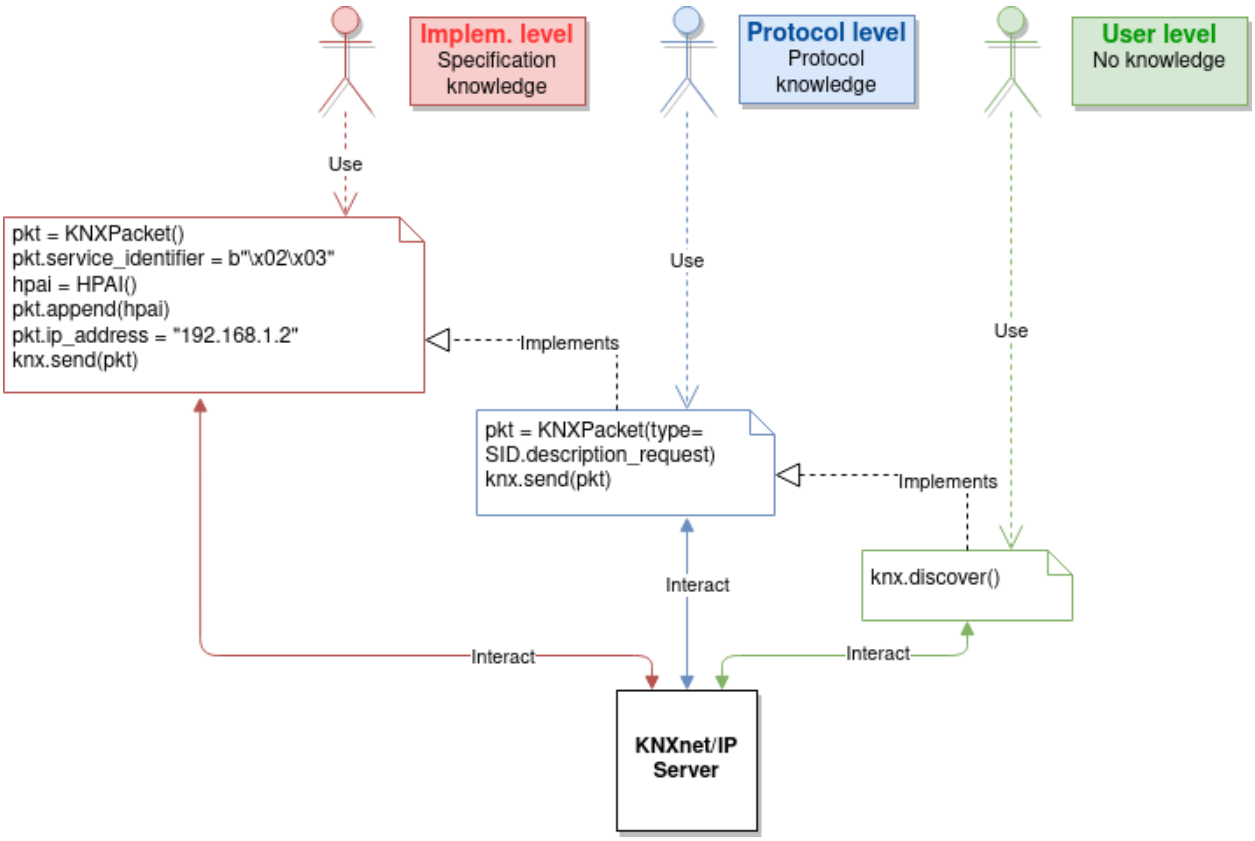
There are three ways to use BOF:

Automated Use of higher-level interaction functions to discover devices and start basic exchanges, without requiring to know anything about the protocol. BOF also has **Modules** that gather these functions.

Standard Perform more advanced (legitimate) operations. This requires the end user to know how the protocol works (how to establish connections, what kind of messages to send).

Playful Modify every single part of exchanged frames and misuse the protocol instead of using it (we fuzz devices with it). The end user should have started digging into the protocol's specifications.

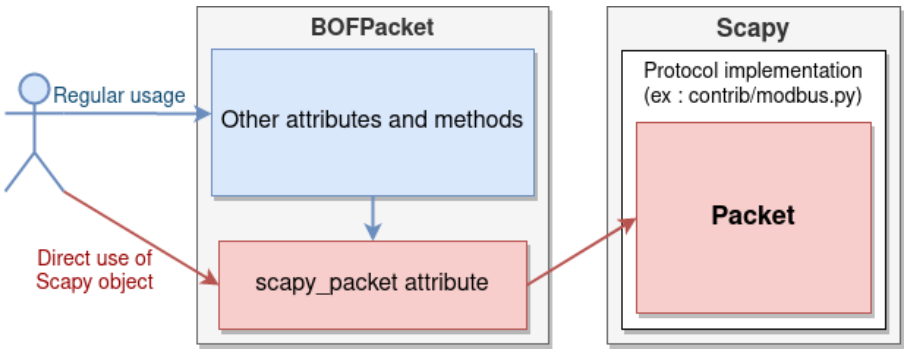
Warning: Please note that targeting industrial systems can have a severe impact on people, industrial operations and buildings and that BOF must be used carefully.



1.2 Interface with Scapy

BOF relies on Scapy for protocol implementations, with an additional layer that translates BOF code to changes on Scapy packets and fields. Why? Because BOF may slightly modify or override Scapy's internal behavior.

You do not need to know how to use Scapy to use BOF, however if you do, you are free to interact with the Scapy packet directly as well.



For instance, in the code sample below, lines 2 and 3 do the same thing and modify the same packet object. However for line 2, you set a value to the `field1` from **BOF**'s packet, applying any change provided by BOF when setting a value. In line 3, the field is modified directly in **Scapy**'s packet, BOF does not interfer. In other words, a `BOFPacket` object (here `KNXPacket`) acts as a wrapper around a Scapy object representing the actual packet using the specified protocol.

```
packet = KNXPacket(type=connect_request)
packet.field1 = 1
packet.scapy_pkt.field1 = 1
```

The reason we did that is because there is nothing better than Scapy to handle protocol implementations, and by using Scapy we can also use all the implementations that were written for it. But BOF and Scapy do not have the same usage and aim. Just to mention a few:

- **Field-oriented usage: BOF's preferred usage when altering packets is to** change specific fields directly. Why? Because BOF has been written to write attack scripts, including fuzzers. In these fuzzers, we want to stick to the protocol's specification because if we don't, devices we target may just drop our frames. But we also want to do whatever we want on packets, sticking to the specification or not. So what we usually do is to modify isolated fields in frames. Scapy does not work this way and, although we can modify fields independently, it's hard to get and set values in a script, mostly because we can't refer to a field without referring to its parent packet holding its value. This also implies that Scapy builds packets as a whole, and performs a final computation / cleaning when building the packet before sending it, and sometimes we don't want that in BOF.
- **BOF does not care about types**: But Scapy does. Field objects in Scapy have a type and you can't change it easily or just use a field object that doesn't have a type without losing some capabilities. For us, packets are just a bunch of bytes so we might as well set values directly as bytes to fields, and Scapy won't allow that (unless using RawVal, which does not provide all of Scapy's Fields capabilities). It won't allow setting a value with the wrong type either, and we don't want field types to be a thing in BOF: a user should not need to know the type of a field, or she may be able to implicitly change it. That's what BOF's wrapper around the Scapy object does.

```
# Setting value to field from BOF, type is changed automatically
bofpacket.host_protocol = "test"

# Setting value to field directly on Scapy packet, type is invalid
# and will trigger an error when the packet is built.
bofpacket.scapy_pkt.control_endpoint.host_protocol = "test"
```

CHAPTER 2

TL;DR

Clone repository:

```
git clone https://github.com/Orange-Cyberdefense/bof.git
```

BOF is a Python 3.6+ library that should be imported in scripts.

```
import bof
```

Global module content can be imported directly from `bof`. Protocol-specific content is in submodule `layers` (ex: `bof.layers.knx`).

```
from bof import BOFProgrammingError
from bof.layers import knx
from bof.layers.knx import *
from bof.modules import discovery
```

Now you can start using BOF!

Note: Examples in this section rely on the protocol KNX, but also apply to the others. Please refer to the **Protocols** section of this documentation for protocol-specific stuff.

2.1 Several ways to discover devices on a network

2.1.1 Passive discovery from the discovery module

```
from bof.modules.discovery import *

devices = passive_discovery(iface="eth0", verbose=True)
```

2.1.2 Device discovery using a layer's high-level function

```
from bof.layers.knx import search

devices = search()
for device in devices:
    print(device)
```

Should output something like:

```
[KNX] Device name: boiboite
      Description: None
      MAC address: 00:00:ff:ff:ff:ff
      IP address: 192.168.1.242
      Port: 3671
      Multicast address: 224.0.23.12
      KNX address: 1.1.1
      Serial number: 0123456789
```

2.1.3 Create and send your own discovery packet

```
from bof.layers.knx import *

pkt = KNXPacket(type="search request")
responses = KNXnet.multicast(pkt, (KNX_MULTICAST_ADDR, KNX_PORT))
for response, _ in responses:
    print(KNXPacket(response))
```

2.2 Send and receive packets

```
from bof.layers.knx import KNXnet, KNXPacket, SID
from bof import BOFNetworkError

try:
    knxnet = KNXnet().connect("192.168.1.242", 3671)
    pkt = KNXPacket(type=SID.description_request,
                    ip_address=knxnet.source_address,
                    port=knxnet.source_port)
    pkt.show2()
    response, _ = knxnet.sr(pkt)
    response.show2()
except BOFNetworkError as bne:
    pass
finally:
    knxnet.disconnect()
```

2.3 Craft your own packets!

```
from bof.layers.knx import KNXPacket, SID
from bof.layers.raw_scapy.knx import LcEMI

pkt = KNXPacket(type=SID.description_request)
pkt.ip_address = b"\x01\x01"
pkt.port = 99999 # Yes it's too large
pkt.append(LcEMI())
pkt.show2() # This may output something strange
```

Note: A recipient device will probably not respond to that, but at least now you know that BOF won't stop you from messing with your packets.

2.4 Basic fuzzing

All `BOFPacket` inheriting packet objects in protocol (e.g. `KNXPacket`) implement a `fuzz()` method.

```
for pkt in KNXPacket(type="configuration request").fuzz():
    knxnet.send(pkt)
```

The method generates packets mutated from the original frame. For each packet, one random field has a random value set. This may not work with all fields depending on their type, and you may also want some fields to remain unchanged. In this case, the `include` or `exclude` arguments can be used.

```
for pkt in base_pkt.fuzz(exclude=("service_identifier")):
    knxnet.send(pkt)
```

CHAPTER 3

Usage

3.1 Getting started with BOF Packets

Important: This section introduces a few general concepts about packet crafting in BOF but does not tell you how to create and manipulate packets with specific protocols. As there may be differences depending on the protocol, please refer to the **Protocols** section for details. Please note that not all layers existing in BOF implement a BOFPacket object.

Protocol-dependent packets you may manipulate in BOF all inherit from `BOFPacket`. For instance, `KNXPacket` is the BOF packet from the protocol KNX. `BOFPacket` is not supposed to be instantiated directly, however it can be useful when you start interacting with unknown/unimplemented protocols.

You can instantiate a packet inheriting from `BOFPacket` as follows:

```
bof_pkt = KNXPacket() # Empty
bof_pkt = KNXPacket(b"\x06\x10"[...]) # From bytes
bof_pkt = KNXPacket(field1=val, field1=val2, etc...) # Set values to fields
```

For `KNX`, packets usually have a `type`, therefore you could do:

```
bof_pkt = KNXPacket(type=SID.description_request)
```

Before going further, you should know that a `BOFPacket` relies on a protocol implementation from Scapy or in Scapy format and will interact with a Scapy `Packet` object relying on this implementation. This implies that:

- There are several features, mostly for printing the content of a frame, inherited from Scapy.
- We have to make a clear distinction between BOF and Scapy content, especially when setting values to fields, hence some usage choices detailed later.
- You can directly use Scapy features, if you interact with `BOFPacket` 's `scapy_pkt` attribute.

3.2 View packets and fields

Here is how to read a complete packet:

```
>>> print(packet)
b'\x06\x10\x02\x03\x00\x0e\x08\x01\x00\x00\x00\x00\x00\x00'

>>> packet.show2()
###[ KNXnet/IP ]###
header_length= 6
protocol_version= 0x10
service_identifier= DESCRIPTION_REQUEST
total_length= 14
###[ DESCRIPTION_REQUEST ]###
     \control_endpoint\
      |###[ HPAI ]###
      |  structure_length= 8
      |  host_protocol= IPV4_UDP
      |  ip_address= 0.0.0.0
      |  port      = 0
```

And to read the value of a field (for instance, `host_protocol`, which is located in the `control_endpoint` PacketField):

```
# Direct access from BOF packet
>>> packet.host_protocol
1

# Reading bytes from BOF packet
>>> packet["host_protocol"]
b'\x01'

# Using BOF packet method get() with no path
>>> packet.get("host_protocol")
1

# Using get() method with absolute or partial path
>>> packet.get("control_endpoint", "host_protocol")
1

# Browsing to Scapy field directly from scapy_pkt attribute
>>> packet.scapy_pkt.control_endpoint.host_protocol
1
```

There are a few things to consider when reaching fields for reading and writing in BOF:

1. `packet.scapy_pkt.host_protocol` won't work, because `scapy_pkt` does not have a `host_protocol` field. It has a `control_endpoint` field which has a `host_protocol`. The complete (absolute) path is required when accessing fields via `scapy_pkt` and not via BOF directly.
2. `packet.control_endpoint.host_protocol` won't work either. If you access fields from BOF, only direct access is allowed (`packet.host_protocol`). This is mainly to avoid confusions between BOF syntax and Scapy syntax (see below). If there are two fields with the same name but different paths in the packet, this syntax will refer to the first one. To refer to a specific one, use `packet.get()`

3.3 Modify packets and fields

BOF does not only set values to packets and fields, it may change Scapy's default behavior when changing the Scapy Packet underneath. The main change is that BOF will replace the field by a field with another type if the value we are trying to set does not match the actual type.

```
>>> type(packet._get_field("host_protocol")[0])
<class 'scapy.fields.ByteEnumField'>
>>> packet.host_protocol = b"hey"
>>> type(packet._get_field("host_protocol")[0])
<class 'scapy.fields.Field'>
```

Therefore, there are two ways of setting a value in BOF.

- The BOF way:

```
>>> packet.host_protocol = b"cor"
>>> packet.host_protocol
b'cor'
>>> packet.update(b"ne", "host_protocol")
>>> packet.host_protocol
b'ne'
>>> packet.update(b"muse", "control_endpoint", "host_protocol")
>>> packet.host_protocol
b'muse'
```

- The Scapy way:

```
>>> packet2.scapy_pkt.control_endpoint.host_protocol = b"nope"
```

The BOF way will set the value while applying changes specific to BOF (ex: replacing a field with a field with a different type). The Packet remains valid (and readable by Scapy's internal features) even if we set the wrong type to a field.

The Scapy way will directly change the value of the Scapy field, BOF will not interfer and will not apply BOF-specific changes. In this last example, we set a value of the wrong type to the field, and an exception will be triggered if you call a method that will try to reconstruct the packet (such as `show2()` or `raw()`).

3.4 Network connection

BOF provides core class for TCP and UDP network connections, however they should not be used directly, but inherited in protocol implementation network connection classes (ex: `KNXnet` inherits `UDP`). A connection class carries information about a network connection and methods to manage connection and exchanges, that can vary depending on the protocol.

Here is an example on how to establish connection using the `knx` submodule (`3671` is the default port for KNXnet/IP).

```
from bof.layers.knx import KNXnet, KNXPacket, SID
from bof import BOFNetworkError

knxnet = KNXnet()
try:
    knxnet.connect("192.168.1.242", 3671)
    pkt = KNXPacket(type=SID.description_request,
                    ip_address=knxnet.source_address,
```

(continues on next page)

(continued from previous page)

```
                    port=knxnet.source_port)
    pkt.show2()
    response, _ = knxnet.sr(pkt)
    response.show2()
except BOFNetworkError as bne:
    pass
finally:
    knxnet.disconnect()
```

There are also various ntework-related functions to use directly. For instance, to send requests via multicast:

```
responses = KNXnet.multicast(pkt, (KNX_MULTICAST_ADDR, KNX_PORT))
```

3.5 Error handling and logging

BOF has custom exceptions inheriting from a global custom exception class `BOFError` (code in *bof/base.py*):

BOFLibraryError Library, files and import-related exceptions.

BOFNetworkError Network-related exceptions (connection errors, etc.).

BOFProgrammingError Misuse of the framework (most frequent one)

```
try:
    knx.connect("invalid", 3671)
except BOFNetworkError as bne:
    print("Connection failure: ".format(str(bne)))

try:
   pkt.KNXPacket(type=SID.configuration_request)
   pkt.update("unknown", 4)
except BOFProgrammingError:
   print("Field does not exist.")
```

Logging features can be enabled for the entire framework. They are disabled by default. Events are stored to a file (default name is `bof.log`). One can make direct call to bof's logger to record custom events.

```
bof.enable_logging()
bof.log("Cannot send data to {0}:{1}".format(ip, port), level="ERROR")
```

CHAPTER 4

Discovery

4.1 Overview

This module constains high-level functions for device discovery on a network using several protocols.

4.2 Passive discovery

When discovering devices on an industrial network, the less we interact directly with devices the better (otherwise we may break something). The `passive_discovery()` function sends identify requests to protocol-specific multicast addresses. Devices that subscribe to them are supposed to respond.

```
passive_discovery(iface="eth0", verbose=True)
```

So far, here is what the function does:

- Listen to **LLDP** multicast address (switches and other network usually send LLDP packets with their description)
- Send a **Profinet DCP** identify request
- Send a **KNXnet/IP** search request

4.3 Other discovery functions

The following discovery functions are available independently:

`lldp_discovery()` Listen on the network for LLDP packets sent on LLDP's multicast MAC address. This function is synchronous. For the async version, call `lldp.start_listening()` and `lldp.stop_listening()`.

`profinet_discovery()` Send an identify request on Profinet DCP's multicast MAC address.

knx_discovery() Send a search request on KNXnet/IP's multicast IP address.

CHAPTER 5

KNX

KNX is a field bus protocol, mainly used for building management systems. BOF implements KNXnet/IP, which is part of the KNX specification to link field KNX components to the IP network.

5.1 Device discovery

BOF provides features to discover devices on a network and gather information about them. Calling them will send the appropriate KNXnet/IP requests to devices and parse their response, you don't need to know how the protocol works.

```
from bof.layers.knx import search

devices = search()
for device in devices:
    print(device)
```

You can also learn more about a specific device:

```
from bof.layers.knx import discover

device = discover("192.168.1.42")
print(device)
```

The resulting object is a `KNXDevice` object that comes with a set of attributes and methods to interact with a device.

Note: The function `knx_discovery()` in the **Discovery** module can also be used (relies on `search()`).

5.2 Send commands

A few commands are available so far to perform basic operations on a KNXnet/IP server or underlying devices:

```
from bof.layers.knx import group_write

# Write value 1 to group address 1/1/1
group_write(device.ip_address, "1/1/1", 1)
```

5.3 Connect to a device

```
from bof.layers import knx
from bof import BOFNetworkError

knxnet = knx.KnxNet()
try:
    knxnet.connect("192.168.1.1", 3671)
    # Do stuff
except BOFNetworkError as bne:
    print(str(bne))
finally:
    knxnet.disconnect()
```

The class `KnxNet` is used to connect to a KNX device (server or object). It creates a UDP connection to a KNX device. `connect` can take an additionnal `init` parameter.

5.4 Send and receive frames

```
from bof.layers.knx import KNXnet, KNXPacket, SID

knxnet = KNXnet().connect("192.168.1.242")
pkt = KNXPacket(type=SID.description_request)
pkt.ip_address, pkt.port = knxnet.source
pkt.show2()
response, _ = knxnet.sr(pkt)
response.show2()
knxnet.disconnect()
```

When a connection is established, one may start sending KNX frames to a device. Frames are sent and received as byte arrays, but they are represented as `KNXPacket` within BOF. In the example above, we create a frame with type `Description Request` to ask a device to describe itself. The format of such frame is extracted from the KNX implementation in Scapy format, either integrated to Scapy or imported to BOF's `raw_scapy` directory. The `response` is received as a byte array, converted to a `KNXPacket` object.

You can also use methods that will directly initialize and send the following basic KNXnet/IP frames.

```
knxnet = KNXnet().connect(ip, port)
# CONNECT REQUEST
channel = connect_request_management(knxnet)
# CONFIGURATION REQUEST with "property read" KNX message
cemi = cemi_property_read(CEMI_OBJECT_TYPES.ip_parameter_object,
                          CEMI_PROPERTIES.pid_additional_individual_addresses)
response = configuration_request(knxnet, channel, cemi)
# DISCONNECT REQUEST
disconnect_request(knxnet, channel)
knxnet.disconnect()
```

Available requests (from KNX Standard v2.1) are:

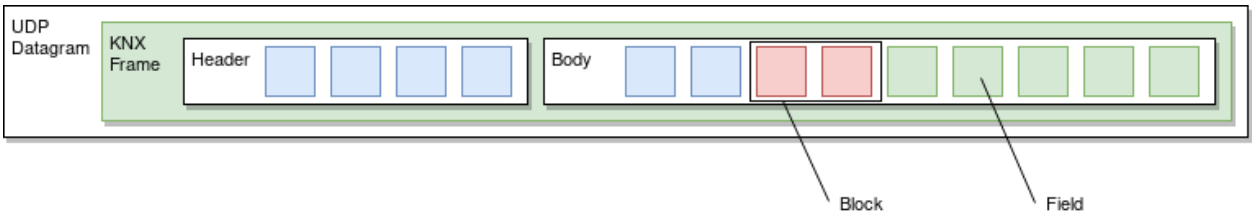
- `Search request`
- `Description request`
- `Connect request` (with connection type "management" and "tunneling")
- `Disconnect request`
- `Configuration request`
- `Tunneling request`

Note: Configuration requests and tunneling requests "carry" medium-independent KNX data in a block called "cEMI". Therefore, when creating such a request you need to specify the type of cEMI to use (see below for details).

5.5 Understanding KNX frames

5.5.1 Structure

Conforming to the KNX Standard v2.1, a KNX frame has a header and body. The header's structure never changes but the body's structure varies according to the type of frame (message) given in the header's `service identifier` field.



A KNX frame contains a set of blocks (set of fields) which contain raw fields or nested block. In BOF (and Scapy), we do not refer to blocks: A `KNXPacket` contains a Scapy `Packet` with `Field` objects. Some `Field` objects act as blocks (yeah, I know...) and may contain other `Field` objects.

5.5.2 Message types

The KNX standard describes a set of message types with different format. Please refer to KNX implementation using Scapy here: `bof/layers/raw_scapy/knx.py` or in Scapy's KNX contrib (should be the same anyway). The header contains a field `service_identifier` that states the type of message. `knx.SID` contains a list of valid types to use when creating a frame:

```
>>> from bof.layers.knx import *
>>> packet = KNXPacket(type=SID.configuration_request)
>>> packet.show2()
###[ KNXnet/IP ]###
  header_length= 6
  protocol_version= 0x10
  service_identifier= CONFIGURATION_REQUEST
  total_length= 21
```

(continues on next page)

(continued from previous page)

```
###[ CONFIGURATION_REQUEST ]###
     structure_length= 4
     communication_channel_id= 1
     sequence_counter= 0
     reserved  = 0
     \cemi      \
      |###[ CEMI ]###
      |  message_code= 0
      |  \cemi_data \
      |   |###[ L_cEMI ]###
      |  [...]
```

Service identifier codes are also directly accepted:

```
>>> packet2 = KNXPacket(type=0x0201)
>>> packet2.show2()
###[ KNXnet/IP ]###
  header_length= 6
  protocol_version= 0x10
  service_identifier= SEARCH_REQUEST
  total_length= 14
###[ ('SEARCH_REQUEST',) ]###
     \discovery_endpoint\
      |###[ HPAI ]###
      |  structure_length= 8
      |  host_protocol= IPV4_UDP
      |  ip_address= 0.0.0.0
      |  port      = 0
```

Specifying no types create an empty KNX Packet.

5.5.3 KNXnet/IP messages vs. KNX messages

We use BOF to interact with a device over IP, that's why we always send KNXnet/IP requests. Some of them stick to "IP" level and will retrieve global information that "exist" at this level (for instance, hardware and network information about a KNXnet/IP server).

```
knx.discover("192.168.1.42")
```

Outputs:

```
Device: "boiboite" @ 192.168.1.242:3671 - KNX address: 15.15.255 -
Hardware: 00:00:ff:ff:ff:ff (SN: 0123456789)
```

However, some requests move to the "KNX" level (the layer below), either to retrieve or send KNX-specific information on a KNXnet/IP server, or to interact with KNX devices underneath. In this case, some KNXnet/IP frames (most notably configuration requests and tunneling requests) will carry a special block containing medium-independent KNX data.

This special KNX data block is called cEMI (for Common External Messaging Interface) and it acts like a frame inside the frame, with its own protocol definition. You can also find it in KNX standard v2.1, but KNXnet/IP specification is not the same as KNX specification.

For instance, "tunneling requests" carry KNX data to be transferred to KNX devices. When you want to write a value to a KNX object, the tunneling request has to carry a specific cEMI message for value write on addresses.

This cEMI message has a type (here, the data link layer message format) and a set of properties of values to indicate what is the expected behavior.

Here is one way to write a KNX write request on a group address with BOF. There are higher-level functions in BOF to do the same thing. For this one you can also just call the `group_write()` function.

```
# Create cEMI block (KNX data)
cemi = scapy_knx.CEMI(message_code=CEMI.l_data_req) # Link layer request
cemi.cemi_data.source_address = knx_source # Retrieved from a connect request
cemi.cemi_data.destination_address = "1/1/1"
cemi.cemi_data.acpi = ACPI.groupvaluewrite # Type of command
cemi.cemi_data.data = value
# Insert it to a tunneling request
tun_req = KNXPacket(type=SID.tunneling_request)
tun_req.communication_channel_id = channel # Retrieved from a connect request
tun_req.cemi = cemi
tun_req.show2()
```

5.6 Testing KNXnet/IP implementations with BOF

BOF provides means to add fields, change their values, even if that does not comply with the protocol. Please refer to the protocol-independent documentation to know how.

Warning: KNX frame servers usually have strict parsing rules and won't consider invalid frames. If you modify the structure of a frame or block and differ too much from the specification, you should not expect the KNX device to respond.

CHAPTER 6

Notice

This section is intended for contributors, either for improving existing parts (core, existing implementation) or adding new protocol implementations. Before going further, please consider the following notice.

6.1 Code quality requirements

Quality We like clean code and expect contributions to be PEP-8 compliant as much as possible (even though we don't test for it). New code should be readable easily and maintainable. And remember: if you need to use "and" while explaining what your function does, then you can probably split it.

Genericity Part of the code (the "core") is used by all protocol implementations. When you add code to the core, please make sure that it does not cause issues in protocol-specific codes. Also, if you write or find out that code in implementations can be made generic and added to the core, feel free to do it.

Unit tests We use Python's `unittest` to write unit tests. When working on BOF, please write or update unit tests! They are in `tests/`. You can run all unit tests with: `python -m unittest discover -s tests`.

6.2 Comments and documentation

Docstrings Modules, functions, classes, methods start with docstrings written in ReStructuredText. Docstrings are extracted to build the ReadTheDocs source code documentation using Sphinx. We use a not-so-strict format, but you should at least make sure that docstrings are useful to the reader, contain the appropriate details and have a valid and consistent format. You can also rely on the following model:

```
"""Brief description of the module, function, class, method.

A few details on how, where, when and why to use it.
```

(continues on next page)

(continued from previous page)

```
:param first: Description of param "first": type, usage, origin
              Second line of description if one isn't enough.
:param second: Description of param "second"
:returns: The value that is returned, if any.
:raises BOFProgrammingError: if misused

Usage example::

  if there is any interest in adding such example, please do so.
"""
```

6.3 Git branching

We follow the "successful git branching model" described here. In a nutshell:

- Branch from `master` for hotfixes
- Work on `dev` for small changes
- Create specific `feature` branches from `dev` for big changes
- Don't work on `master`

6.4 Report issues

Report bugs, ask questions or request for missing documentation and new features by submitting an issue on GitHub. For bugs, please describe your problem as clearly as you can.

CHAPTER 7

Architecture

The library has the following structure:

```
.
├── bof
│   ├── base.py
│   ├── network.py
│   ├── packet.py
│   ├── __init__.py
│   ├── layers
│   │   ├── knx
│   │   │   ├── knx_constants.py
│   │   │   ├── knx_device.py
│   │   │   ├── knx_functions.py
│   │   │   ├── knx_network.py
│   │   │   └── knx_packet.py
│   │   ├── other protocol
│   │   │   └── other protocol content
│   │   └── raw_scapy
│   │       └── knx.py
│   ├── modules
│   │   └── discovery.py
```

- The protocol-independent part of BOF (the core) is in `bof` directly.
- BOF protocol features are in `bof/layers/[protocol]`
- Scapy protocol implementations are imported directly from Scapy or can be stored in `bof/layers/raw_scapy/[protocol].py`
- Higher-level functions not speciic to a layer are in `modules` (e.g. the `discovery` module for device discovery on a network using several protocols).

Apart from the library:

- The documentation as RestructuredText files for Sphinx is in `docs`
- Unit tests (one file for the core, one file per protocol) are in `tests`

- Implementation examples are in `examples/[protocol|module]`

CHAPTER 8

Extend BOF

Here is how to add a new protocol to BOF:

1. Make sure that the protocol exist in Scapy or provide an implementation in Scapy format (the file can be stored in `bof/layers/raw_scapy`).
2. Create a folder in `bof/layers` with the name of your implementation. Here we'll add the protocol `otter`.
3. In `bof/layers/otter`, create a Python file with a class inerithing either from TCP or UDP (they are in `bof/network.py`). It will contain any protocol-related operations at network level. For instance, you may overwrite send and receive operation so that they return `OtterPacket` directly.
4. Create another Python file to write a class `OtterPacket` (or whatever) inheriting from `BOFPacket`.

```
class OtterPacket(BOFPacket):
```

5. Please refer to `BOFPacket` (in `bof/packet.py`) and to other implementations such as `KNX` to know how to write the content of the class, until I write a better tutorial! :D
6. Additionnaly, you can create a Python file to write higher-level functions (for instance, objects inheriting `BOFDevice` and functions that creates it), and move your protocol-dependent constants to a dedicated Python file.

Note: You can also create the layer with only higher-level functions that rely directly on the Scapy packet without BOF's overrides (i.e.: no BOFPacket object). Layers LLDP and Profinet currently work this way.

CHAPTER 9

Introduction

Boiboite Opener Framework / Ouvre-Boiboite Framework contains a set of features to write scripts using industrial network protocols for test and attack purposes.

The following submodules are available:

base Basic helpers for correct module usage (error handling, logging, some parsing features.

network Global network classes, used by protocol implementations in submodules. The content of this class should not be used directly, unless writing a new protocol submodule.

packet Base class for specialized BOF packets in layers. Such classes link BOF content and usage to protocol implementations in Scapy. In other words, they interface BOF's syntax used by the end user with Scapy Packet and Field objects used for the packet itself. The base class `BOFPacket` is not supposed to be instantiated directly, but whatever.

device Global object for representing industrial devices. All objects in layers built using data extracted from responses to protocol-specific discovery requests shall inherit `BOFDevice`.

layers Protocol implementations to be imported in BOF. Importing `layers` gives access to BOF protocol implementations inheriting from `BOFPacket` (interface between BOF and Scapy worlds). The directory `layers/raw_scapy` may contain protocol implementations in Scapy which are not integrated to Scapy's repository (for instance, if you wrote your own but did not contribute (yet)).

modules Higher level functions gathered around a specific usage that may rely on several protocols (layers).

CHAPTER 10

Basic and global functions

10.1 Global settings and error handling

Set of global and useful classes and functions used within the module.

Exceptions BOF-specific exceptions raised by the module.

Logging Functions to enable or disable logging for the module.

String manipulation Functions to make basic changes on strings.

exception bof.base.**BOFError**
Bases: Exception

Base class for all BOF exceptions.

> **Warning:** Should not be used directly, please raise or catch subclasses instead.

exception bof.base.**BOFLibraryError**
Bases: *bof.base.BOFError*

Library, files and import-related exceptions.

Raise when the library cannot find what it needs to work correctly (such as an external module or a file).

exception bof.base.**BOFNetworkError**
Bases: *bof.base.BOFError*

Network-related exceptions.

Raise when the network connection fails or is interrupted.

exception bof.base.**BOFProgrammingError**
Bases: *bof.base.BOFError*

Script and module programming-related errors.

Raise when a function or an argument is not used as expected.

Note: As a module user, this exception is the most frequent one.

bof.base.**disable_logging**() → None
Turn off logging features,

bof.base.**enable_logging**(*filename: str = ''*, *error_only: bool = False*) → None
Turn on logging features to store BOF-autogenerated and user events. Relies on Python's `logging` module.

Parameters

- **filename** – Optional name of the file in which events will be saved. Default is `bof.log`.
- **error_only** – All types of events are logged (info, warning, error) are saved unless this parameter is set to `True`.

bof.base.**log**(*message: str*, *level: str = 'INFO'*) → bool
Logs an event (`message`) to a file, if BOF logging is enabled. Requires previous call to *bof.'enable_logging()'*. A `message` is recorded along with event-related information:

- date and time
- level (can be changed with parameter `level`)
- event location in the code (file name, line number)

Parameters

- **message** – Event definition.
- **level** – Type of event to record: `ERROR`, `WARNING`, `DEBUG`. *INFO'* (default). Levels from Python's `logging` are used.

Returns Current state of logging (enabled/`True`, disabled/`False`).

bof.base.**to_property**(*value: str*) → str
Lower a string and replace all non alnum characters with _

10.2 Basic network protocol implementation

Network protocol global classes and abstract implementations.

Provides classes for asynchronous network connection management on different transport protocols, to be used by higher-level protocol implementation classes. Relies on module `asyncio`.

UDP Implementation of asynchronous UDP communication and packet crafting.

TCP Implementation of asynchronous TCP communication and packet crafting.

Both classes rely on internal class `_Transport`, which should not be instantiated.

Network connection and exchange example with raw UDP:

```
from bof import UDP
udp = UDP()
udp.connect("192.168.1.1", 3671)
udp.send(b"Hi!")
udp.disconnect()
```

Usage is the same with raw TCP.

> **Warning:** Direct initialization of TCP/UDP object is not recommended. The user should use BOF network classes inherited from TCP/UDP (e.g. `KNXnet` for the `KNX` protocol).

`bof.network.`**`IS_IP`**(*ip: str*)
Check that ip is a valid IPv4 address.

class `bof.network.`**`TCP`**
Bases: `bof.network._Transport`

TCP protocol endpoint.

This is the parent class to higher-lever network protocol implementation. It can be instantiated as is, however this is not the expected behavior. Uses protected `_TCP` classes implementing `asyncio` TCP handler.

> **Warning:** Should not be instantiated directly.

`connect`(*ip: str*, *port: int*) → object
Initialize asynchronous connection using TCP on `ip:port`.

Parameters

- **`ip`** – IPv4 address as a string with format `A.B.C.D`.
- **`port`** – Port number as an integer.

Returns The instance of the TCP class created,

Raises ***`BOFNetworkError`*** – if connection fails.

Example:

```
tcp = bof.TCP().connect("127.0.0.1", 4840)
```

`send`(*data: bytes*, *address: tuple = None*) → int
Send `data` to `address` over TCP.

Parameters

- **`data`** – Raw byte array or string to send.
- **`address`** – Address to send `data` to, with format tuple `(ipv4_address, port)`. If address is not specified, uses the address given to `connect`.

Returns The number of bytes sent, as an integer.

Example:

```
tcp.send("test_send")
tcp.send(b'}')
```

class `bof.network.`**`UDP`**
Bases: `bof.network._Transport`

UDP protocol endpoint, inheriting from Transport base class.

This is the parent class to higher-lever network protocol implementation. It can be instantiated as is, however this is not the expected behavior. Uses protected `_UDP` classes implementing `asyncio` UDP handler.

> **Warning:** Should not be instantiated directly.

static broadcast (*data: bytes*, *address: tuple*, *timeout: float = 1.0*) → list
Broadcasts a request and waits for responses from devices (UDP).

Parameters

- **data** – Raw byte array or string to send.
- **address** – Remote network address with format tuple `(ip, port)`.
- **timeout** – Time out value in seconds, as a float (default is 1.0s).

Returns A list of tuples with format `(response, (ip, port))`.

Raises ***BOFNetworkError*** – If multicast parameters are invalid.

Example:

```
devices = UDP.broadcast(b'...', ('192.168.1.255', 3671))
```

connect (*ip: str*, *port: int*) → object
Initialize asynchronous connection using UDP on `ip`:`port`.

Parameters

- **ip** – IPv4 address as a string with format `A.B.C.D`.
- **port** – Port number as an integer.

Returns The instance of the UDP class created,

Raises ***BOFNetworkError*** – if connection fails.

Example:

```
udp = bof.UDP().connect("127.0.0.1", 13671)
```

static multicast (*data: bytes*, *address: tuple*, *timeout: float = 1.0*) → list
Sends a multicast request to specified ip address and port (UDP).

Expects devices subscribed to the address to respond and return responses as a list of frames with their source. Opens its own socket.

Parameters

- **data** – Raw byte array or string to send.
- **address** – Remote network address with format tuple `(ip, port)`.
- **timeout** – Time out value in seconds, as a float (default is 1.0s).

Returns A list of tuples with format `(response, (ip, port))`.

Raises ***BOFNetworkError*** – If multicast parameters are invalid.

Example:

```
devices = UDP.multicast(b'...', ('224.0.23.12', 3671))
```

send (*data: bytes*, *address: tuple = None*) → int
Send `data` to `address` over UDP.

Parameters

- **data** – Raw byte array or string to send.
- **address** – Address to send `data` to, with format tuple `(ipv4_address, port)`. If address is not specified, uses the address given to `connect`.

Returns The number of bytes sent, as an integer.

Example:

```
udp.send("test_send")
udp.send(b'}')
```

10.3 BOFPacket base class

Interfaces with a packet as a Scapy object, with specific features.

A BOFPacket is a sort of wrapper around a Scapy Packet object, and implements specific features or changes relative to Scapy's behavior when interacting with this packet.

The Scapy Packet is used as a basis for BOF to manipulate frames with its own syntax. You don't need to know how to use Scapy to use BOF. However, you can still perform "Scapy stuff" on the packet by directly accessing `BOFPacket.scapy_pkt` attribute.

Note: BOFPacket DOES NOT inherit from Scapy packet, because we don't need a "specialized" class, but a "translation" from BOF usage to Scapy objects.

Example (keep in mind that BOFPacket should not be instantiated directly :)):

```
pkt = BOFPacket(scapy_pkt=ScapyBasicOtterPacket1())
print(pkt.scapy_pkt.basic_otter_1_1, pkt.basic_otter_1_1) # Same output
pkt.basic_otter_1_1 = "192.168.1.2" # Not the expected type, BOF converts it
pkt.show2()
```

class bof.packet.**BOFPacket**(*_pkt: bytes = None*, *scapy_pkt: scapy.packet.Packet = None*, ***kwargs*)

Bases: `object`

Base class for BOF network packet handling, to inherit in subclasses.

This class should not be instantiated directly but protocol-specific Packet classes in BOF shall inherit it. It acts as a wrapper around Scapy-based packets in the specified protocol, either relaying, replacing or modifying Scapy default behaviors on Packets and Fields.

Parameters

- **_pkt** – Raw Packet bytes used to build a packet (mostly done at reception, but you can manually create a packet from bytes)
- **scapy_pkt** – Actual Scapy `Packet` object, used by BOF for protocol implementation-related stuff. Can be referred to directly to do "Scapy stuff" inside BOF.
- **kwargs** – Field values to set when instantiating the class. Format is `field_name=value, ...`. If two fields have the same name, it sets the first one.

Example:

```
class OtterPacket(BOFPacket)
```

append(*other: object*, *autobind: bool = False*, *packet=None*, *value=None*) → None
Adds either a BOFPacket, Scapy Packet or Field to current packet.

Parameters

- **other** – BOFPacket or Scapy Packet or field to append as payload.
- **autobind** – Whether or not unspecified binding found in Scapy implementation are automatically added.
- **packet** – Packet at to append `other` to.
- **value** – Value to set to a newly-created field.

Raises ***BOFProgrammingError*** – if type is not supported.

copy()
Copies the current instance by rebuilding it from its bytes. Works appropriately only if the original packet is valid. Any attribute not strictly bound to bytes is ignored, you should add it.

Example:

```
copy_of_pkt = self.copy()
copy_of_pkt.show2() # Should be the same thing as self.show2()
```

fields
Returns the list of field objects in a `BOFPacket`.

Can be used to retrieve the list of fields as a name list with:

```
[x.name for x in pkt.fields]
```

fuzz(*iterations: int = 0*, *include: list = None*, *exclude: list = None*)
Generator function. Sets a random value to a random field in packet.

Parameters

- **iterations** – Number of packet to create (default is infinite loop)
- **include** – List of field names to include to fuzzing.
- **exclude** – List of field names to exclude from fuzzing.

Example:

```
pkt = KNXPacket(type="configuration request")
for frame in pkt.fuzz():
  print(frame)
```

get(**args*) → object
Get a field either from its name, partial or absolute path.

Partial indicates part of the absolute path, in other words where the search for the field should start from.

Parameters **args** – Can take from one to many arguments. The last argument must be the field you look for. Previous "path" arguments must be in the right order (even if the path is not complete).

Raises ***BOFProgrammingError*** – If field not found or not supported.

length
Returns the length of the packet (number of bytes).

scapy_pkt

type
Get information about the packet's type (protocol-dependent).

Should be overriden in subclasses to match a protocol's different types of packets. For instance, BOF's packet for the KNX protocol (`KNXPacket`) returns the type of packet as a name, relying on its identifier fields. If identifier is 0x0203, `pkt.type` indicates that the packet is a `DESCRIPTION REQUEST`.

update(*value: object*, **args*) → None
Set value to a field either from its name, partial or absolute path.

Partial indicates part of the absolute path, in other words where the search for the field should start from.

Parameters

- **value** – The value to set to the field. If the type does not match, the type of field will be changed.
- **args** – Can take from one to many arguments. The last argument must be the field you look for. Previous "path" arguments must be in the right order (even if the path is not complete).

Raises ***BOFProgrammingError*** – If field not found or not supported.

10.4 BOFDevice base class

Global object for representing industrial devices.

All objects in layers built using data extracted from responses to protocol-specific discovery requests shall inherit `BOFDevice`.

class bof.device.**BOFDevice**(*name: str = None*, *description: str = None*, *mac_address: str = None*, *ip_address: str = None*)
Bases: `object`

Interface class for devices, to inherit in layer-specific device classes.

Device objects are usually built from device description requests in layers. A device has a set of basic information: a name, a description, a MAC address and an IP address. All of them are attributes to this base object, but not all of them may be provided when asking protocols for device descriptions. On the other hand, most of protocol-specific devices will have additional attributes.

description = None

ip_address = None

mac_address = None

name = None

protocol = 'BOF'

CHAPTER 11

Modules

11.1 Using modules

Modules are higher-level features provided by BOF. They can rely on one or more layer, depending on what they do. Basically, each module is a collection of functions to call in a script.

List of modules:

- **Discovery**: Functions to gather initial information on industrial devices on a network, using active and passive techniques. Rely on several protocols.

11.2 Discovery

11.2.1 Module: Discovery

Functions for passive and active discovery of industrial devices on a network.

`bof.modules.discovery.`**`knx_discovery`**(*ip: str = '224.0.23.12'*, *port=3671*, ***kwargs*)
Search for KNX devices on an network using multicast.

Implementation in KNX layer.

`bof.modules.discovery.`**`lldp_discovery`**(*iface: str = 'eth0'*, *timeout: int = 20*) → list
Search for devices on an network by listening to LLDP requests.

Converts back asynchronous to synchronous with sleep (silly I know). If you want to keep asynchrone, call directly `start_listening` and `stop_listening` in your code.

Implementation in LLDP layer.

`bof.modules.discovery.`**`passive_discovery`**(*iface: str = 'eth0'*, *pndcp_multicast: str = '01:0e:cf:00:00:00'*, *knx_multicast: str = '224.0.23.12'*, *verbose: bool = False*)
Discover devices on an industrial network using passive methods.

Requests are sent to protocols' multicast addresses or via broadcast. Currently, LLDP and KNX are supported.

Parameters

- **lldp_multicast** – Multicast MAC address for LLDP requests.
- **knx_multicast** – Multicast IP address for KNXnet/IP requests.

bof.modules.discovery.**profinet_discovery**(*iface: str = 'eth0'*, *mac_addr: str = '01:0e:cf:00:00:00'*) → list

Search for devices on an network using multicast Profinet DCP requests.

Implementation in Profinet layer.

CHAPTER 12

Layers

12.1 Using layers

BOF relies on protocol implementations built using the Scapy syntax, to provide security testing and fuzzing features. In other words, BOF works as follows:

The `layers` folder contain BOF features for implemented protocols.

Scapy protocol implementations can be imported directly from Scapy or from a KNX implementation not integrated to Scapy that should be located in the `layers/raw_scapy` folder.

12.2 KNX

12.2.1 KNX and KNXnet/IP

KNX is a common field bus protocol in Europe, mostly used in Building Management Systems. KNXnet/IP is the version of the protocol over IP, implementing specific type of frames that either ask information from or send request to a gateway (server) between an IP network and a KNX bus or carry KNX messages that the gateway must relay to KNX devieces on the field bus.

The protocol is a merge a several older ones, the specifications are maintained by the KNX association and can be found on their website (section 3 is the interesting one).

BOF's `knx` submodule can be imported with:

```
from bof.layers import knx
from bof.layers.knx import *
```

The following files are available in the module:

knx_network Class for network communication with KNX over UDP. Inherits from BOF's `network UDP` class. Implements methods to connect, disconnect and mostly send and receive frames as `KNXPacket` objects.

knx_packet Object representation of a KNX packet. `KNXPacket` inherits `BOFPacket` and uses Scapy's implementation of KNX (located in `bof/layers/raw_scapy` or directly in Scapy contrib). Contains method to build, read or alter a frame or part of it, even if this does not follow KNX's specifications.

knx_messages Set of functions that build specific KNX messages with the right values.

knx_functions Higher-level functions to discover and interact with devices via KNXnet/IP.

12.2.2 Network connection

KNXnet/IP connection features, implementing `bof.network`'s `UDP` class.

The `KnxNet` class translates `KNXPacket` packet objects and raw Scapy packets to bytes to send them, and received bytes to `KNXPacket` objects.

KNX usually works over UDP, however KNX specification v2.1 state that TCP can also be used. The communication between BOF and a KNX device still acts like a TCP-based protocol, as (almost) every request expects a response.

Usage:

```
knxnet = KNXnet()
knxnet.connect("192.168.1.242")
data, addr = knxnet.sr(KNXPacket(type=SID.description_request))
data.show2()
knxnet.disconnect()
```

class bof.layers.knx.knx_network.**KNXnet**

Bases: *bof.network.UDP*

KNXnet/IP communication over UDP with protocol KNX. Relies on `bof.network.UDP()`.

Sent and received datagrams are returned as `KNXPacket()` objects.

..seealso:: Details on data exchange: **KNX Standard v2.1 - 03_03_04**.

connect(*ip: str*, *port: int = 3671*) → object

Connect to a KNX device (opens socket). Default port is `3671`.

Parameters

- **ip** – IPv4 address as a string with format `A.B.C.D`.
- **port** – KNX port. Default is `3671`.

Returns The KNXnet connection object (this instance).

Raises ***BOFNetworkError*** – if connection fails.

receive(*timeout: float = 1.0*) → object

Converts received bytes to a parsed `KNXPacket` object.

Parameters **timeout** – Time to wait to receive a frame (default is 1 sec)

Returns A `KNXPacket` object.

send(*data: object*, *address: tuple = None*) → int

Converts BOF and Scapy frames to bytes to send. Relies on `UDP` class to send data.

Parameters

- **data** – Data to send as `KNXPacket`, Scapy `Packet`, string or bytes. Will be converted to bytes anyway.

- **address** – Address to send `data` to, with format `(ip, port)`. If address is not specified, uses the address given to ‘‘ connect‘‘.

Returns The number of bytes sent, as an integer.

sequence_counter = None

12.2.3 KNXPacket

This class inheriting from `BOFPacket` is the interface between BOF's usage of KNX by the end user and an actual Scapy packet built using KNX's implementation in Scapy format.

In BOFPacket and KNXPacket, several builtin methods and attributes are just relayed to the Scapy Packet underneath. We also want to let the user interact directly with the Scapy packet if she wants, using `scapy_pkt` attribute.

Example:

```
>>> from bof.layers.knx import *
>>> packet = KNXPacket(type=SID.description_request)
>>> packet
<bof.layers.knx.knx_packet.KNXPacket object at 0x7ff74224add8>
>>> packet.scapy_pkt
<KNX  service_identifier=DESCRIPTION_REQUEST |<KNXDescriptionRequest      control_
↪endpoint=<HPAI  |> |>>
```

class bof.layers.knx.knx_packet.**KNXPacket**(*_pkt: bytes = None, scapy_pkt: scapy.packet.Packet = None, type: object = None, **kwargs*)

Bases: *bof.packet.BOFPacket*

Builds a KNXPacket packet from a byte array or from attributes.

Parameters

- **_pkt** – KNX frame as byte array to build KNXPacket from.
- **scapy_pkt** – Instantiated Scapy Packet to use as a KNXPacket.
- **type** – Type of frame to build. Ignored if `_pkt` set. Should be a value from `SID` dict imported from KNX Scapy implementation as a dict key, a string or as bytes.
- **kwargs** – Any field to initialize when instantiating the frame, with format field_name=value.

Example of initialization:

```
pkt = KNXPacket(b"[...]") # From frame as a byte array
pkt = KNXPacket(type=SID.description_request) # From service id dict
pkt = KNXPacket(type="DESCRIPTION REQUEST") # From service id name
pkt = KNXPacket(type=b"}") # From service id value
pkt = KNXPacket(type=SID.connect_request, communication_channel_id=2)
pkt = KNXPacket(scapy_pkt=KNX()/KNXDescriptionRequest()) # With Scapy Packet
pkt = KNXPacket() # Empty packet (just a KNX header)
```

set_type(*ptype: object*, *cemi: object = None*) → None

Format packet according to the specified type (service identifier).

Parameters

- **ptype** – Type of frame to build. Ignored if `_pkt` set. Should be a value from `SID` dict imported from KNX Scapy implementation as a dict key, a string or as bytes.

- **cemi** – cEMI field type. Raises error if type does not have have a cEMI field, is ignored if there is no type given.

Raises *BOFProgrammingError* – if type is unknown or invalid or if cEMI is set but there is no cEMI field in packet type.

sid

type

Get information about the packet's type (protocol-dependent).

Should be overriden in subclasses to match a protocol's different types of packets. For instance, BOF's packet for the KNX protocol (`KNXPacket`) returns the type of packet as a name, relying on its identifier fields. If identifier is 0x0203, `pkt.type` indicates that the packet is a `DESCRIPTION REQUEST`.

12.2.4 KNX messages

Module containing a set of functions to build predefined types of KNX messages. Functions in this module do not handle the network exchange, they just create ready-to-send packets.

Contents:

KNXnet/IP requests Direct methods to create initialized requests from the standard.

CEMI Methods to create specific type of cEMI messages (protocol-independent KNX messages).

bof.layers.knx.knx_messages.**cemi_ack**(*knx_indiv_addr: str*, *seq_num: int = 0*, *knx_source: str = '0.0.0'*) → scapy.packet.Packet

Builds a KNX message (cEMI) to disconnect from an individual address.

Parameters

- **knx_indiv_addr** – KNX individual address of device (with format X.Y.Z)
- **seq_num** – Sequence number to use, applies to cEMI when sequence_type is set to "numbered". So far I haven't seen seq_num > 0.
- **knx_source** – KNX individual address to use as a source for the request. You should usually use the KNXnet/IP server's individual address, but it works fine with 0.0.0.

Returns A raw cEMI object from Scapy's implementation to be inserted in a KNXPacket object.

Raises *BOFProgrammingError* – if KNX addresses are invalid because the Scapy object does not allow that. You should change the field type if you want to set somethig else.

bof.layers.knx.knx_messages.**cemi_connect**(*knx_indiv_addr: str*, *knx_source: str = '0.0.0'*) → scapy.packet.Packet

Builds a KNX message (cEMI) to connect to an individual address.

Parameters

- **knx_indiv_addr** – KNX individual address of device (with format X.Y.Z)
- **knx_source** – KNX individual address to use as a source for the request. You should usually use the KNXnet/IP server's individual address, but it works fine with 0.0.0.

Returns A raw cEMI object from Scapy's implementation to be inserted in a KNXPacket object.

Raises *BOFProgrammingError* – if KNX addresses are invalid because the Scapy object does not allow that. You should change the field type if you want to set somethig else.

bof.layers.knx.knx_messages.**cemi_dev_descr_read**(*knx_indiv_addr: str*, *seq_num: int = 0*, *knx_source: str = '0.0.0'*) → scapy.packet.Packet

Builds a KNX message (cEMI) to write a value to a group address.

Parameters

- **knx_indiv_addr** – KNX individual address of device (with format X.Y.Z)
- **seq_num** – Sequence number to use, applies to cEMI when sequence_type is set to "numbered". So far I haven't seen seq_num > 0.
- **knx_source** – KNX individual address to use as a source for the request. You should usually use the KNXnet/IP server's individual address, but it works fine with 0.0.0.

Returns A raw cEMI object from Scapy's implementation to be inserted in a KNXPacket object.

Raises ***BOFProgrammingError*** – if KNX addresses are invalid because the Scapy object does not allow that. You should change the field type if you want to set somethig else.

bof.layers.knx.knx_messages.**cemi_disconnect**(*knx_indiv_addr: str*, *knx_source: str = '0.0.0'*) → scapy.packet.Packet

Builds a KNX message (cEMI) to disconnect from an individual address.

Parameters

- **knx_indiv_addr** – KNX individual address of device (with format X.Y.Z)
- **knx_source** – KNX individual address to use as a source for the request. You should usually use the KNXnet/IP server's individual address, but it works fine with 0.0.0.

Returns A raw cEMI object from Scapy's implementation to be inserted in a KNXPacket object.

Raises ***BOFProgrammingError*** – if KNX addresses are invalid because the Scapy object does not allow that. You should change the field type if you want to set somethig else.

bof.layers.knx.knx_messages.**cemi_group_write**(*knx_group_addr: str*, *value*, *knx_source: str = '0.0.0'*) → scapy.packet.Packet

Builds a KNX message (cEMI) to write a value to a group address.

Parameters

- **knx_group_addr** – KNX group address targeted (with format X/Y/Z) Group addresses are defined in KNX project settings.
- **value** – Value to set the group address' content to.
- **knx_source** – KNX individual address to use as a source for the request. You should usually use the KNXnet/IP server's individual address, but it works fine with 0.0.0.

Returns A raw cEMI object from Scapy's implementation to be inserted in a KNXPacket object.

Raises ***BOFProgrammingError*** – if KNX addresses are invalid because the Scapy object does not allow that. You should change the field type if you want to set somethig else.

bof.layers.knx.knx_messages.**cemi_property_read**(*object_type: int*, *property_id: int*) → scapy.packet.Packet

Builds a KNX message (cEMI) to write a value to a group address.

Parameters

- **object_type** – Type of object to read, as defined in KNX Standard (and reproduce in Scapy's KNX implementation).
- **property_id** – Property to read, as defined in KNX Standard (and reproduce in Scapy's KNX implementation).

Returns A raw cEMI object from Scapy's implementation to be inserted in a KNXPacket object.

bof.layers.knx.knx_messages.**configuration_ack**(*channel: int*) → bof.layers.knx.knx_packet.KNXPacket

Creates a configuration ack to reply to avoid upsetting KNX servers.

bof.layers.knx.knx_messages.**configuration_request**(*channel: int*, *cemi: scapy.packet.Packet*) → bof.layers.knx.knx_packet.KNXPacket

Creates a configuration request with a specified cEMI message.

Parameters

- **channel** – The communication channel ID for the current KNXnet/IP connection. The channel is set by the server and returned in connect responses.
- **cemi** – Protocol-independent KNX message inserted in the request. cEMI are created directly from Scapy's CEMI object.

Returns A configuration request embedding a cEMI packet, as a KNXPacket.

bof.layers.knx.knx_messages.**connect_request_management**(*knxnet: bof.layers.knx.knx_network.KNXnet = None*) → bof.layers.knx.knx_packet.KNXPacket

Creates a connect request with device management connection type.

Parameters **knxnet** – The KNXnet connection object to use. We only need the source parameter, please create an issue if you think that asking directly for the source instead is a better choice.

Returns A management connect request as a KNXPacket.

bof.layers.knx.knx_messages.**connect_request_tunneling**(*knxnet: bof.layers.knx.knx_network.KNXnet = None*) → bof.layers.knx.knx_packet.KNXPacket

Creates a connect request with tunneling connection type.

Parameters **knxnet** – The KNXnet connection object to use. We only need the source parameter, please create an issue if you think that asking directly for the source instead is a better choice.

Returns A tunneling connect request as a KNXPacket.

bof.layers.knx.knx_messages.**description_request**(*knxnet: bof.layers.knx.knx_network.KNXnet = None*) → bof.layers.knx.knx_packet.KNXPacket

Creates a basic description request with appropriate source.

Parameters **knxnet** – The KNXnet connection object to use. We only need the source parameter, please create an issue if you think that asking directly for the source instead is a better choice.

Returns A description request as a KNXPacket.

bof.layers.knx.knx_messages.**disconnect_request**(*knxnet: bof.layers.knx.knx_network.KNXnet = None*, *channel: int = 1*) → bof.layers.knx.knx_packet.KNXPacket

Creates a disconnect request to close connection on given channel.

Parameters

- **knxnet** – The KNXnet connection object to use. We only need the source parameter, please create an issue if you think that asking directly for the source instead is a better choice.

- **channel** – The communication channel ID for the current KNXnet/IP connection. The channel is set by the server and returned in connect responses.

Returns A disconnect request as a KNXPacket.

bof.layers.knx.knx_messages.**search_request**(*knxnet: bof.layers.knx.knx_network.KNXnet = None*) → bof.layers.knx.knx_packet.KNXPacket

Creates a basic search request with appropriate source.

Parameters **knxnet** – The KNXnet connection object to use. We only need the source parameter, please create an issue if you think that asking directly for the source instead is a better choice.

Returns A search request as a KNXPacket.

bof.layers.knx.knx_messages.**tunneling_ack**(*channel: int*, *sequence_counter: int*) → bof.layers.knx.knx_packet.KNXPacket

Creates a tunneling ack to reply to avoid upsetting KNX servers.

bof.layers.knx.knx_messages.**tunneling_request**(*channel: int*, *sequence_counter: int*, *cemi: scapy.packet.Packet*) → bof.layers.knx.knx_packet.KNXPacket

Creates a tunneling request with a specified cEMI message.

Parameters

- **channel** – The communication channel ID for the current KNXnet/IP connection. The channel is set by the server and returned in connect responses.
- **sequence_counter** – Sequence number to use for the request, same principle as TCP's sequence numbers.
- **cemi** – Protocol-independent KNX message inserted in the request. cEMI are created directly from Scapy's CEMI object.

Returns A tunneling request embedding a cEMI packet, as a KNXPacket.

12.2.5 KNX functions

Higher-level functions to interact with devices using KNXnet/IP.

Contents:

KNXDevice Object representation of a KNX device with multiple properties. Only supports KNXnet/IP servers so far, but will be extended to KNX devices.

Functions High-level functions to interact with a device: search, discover, read, write, etc.

Relies on **KNX Standard v2.1**

bof.layers.knx.knx_functions.**GROUP_ADDR**(*x: int*) → str
Converts an int to KNX group address.

bof.layers.knx.knx_functions.**INDIV_ADDR**(*x: int*) → str
Converts an int to KNX individual address.

class bof.layers.knx.knx_functions.**KNXDevice**(*name: str*, *ip_address: str*, *port: int*, *knx_address: str*, *mac_address: str*, *multicast_address: str = '224.0.23.12'*, *serial_number: str = ''*)

Bases: *bof.device.BOFDevice*

Object representing a KNX device.

Data stored to the object is the one returned by SEARCH RESPONSE and DESCRIPTION RESPONSE messages, stored to public attributes:

```
Device name, IPv4 address, KNXnet/IP port, KNX individual address, MAC
address, KNX multicast address used, device serial number.
```

This class provides two factory class methods to build a KNXDevice object from search responses and description responses.

The information gathered from devices may be completed, improved later.

classmethod init_from_description_response(*response: bof.layers.knx.knx_packet.KNXPacket*, *source: tuple*)

Set appropriate values according to the content of description response.

Parameters

- **response** – Description Response provided by a device as a KNXPacket.
- **source** – Source of the response, usually provided in KNXnet's receive() and sr() return values.

Returns A KNXDevice object.

Usage example:

```
response, source = knxnet.sr(description_request(knxnet))
device = KNXDevice.init_from_description_response(response, source)
```

classmethod init_from_search_response(*response: bof.layers.knx.knx_packet.KNXPacket*)

Set appropriate values according to the content of search response.

Parameters **response** – Search Response provided by a device as a KNXPacket.

Returns A KNXDevice object.

Uage example:

```
responses = KNXnet.multicast(search_request(), (ip, port))
for response, source in responses:
  device = KNXDevice.init_from_search_response(KNXPacket(response))
```

protocol = 'KNX'

bof.layers.knx.knx_functions.**discover**(*ip: str*, *port: int = 3671*) → bof.layers.knx.knx_functions.KNXDevice

Returns discovered information about a device. So far, only sends a DESCRIPTION REQUEST and uses the DESCRIPTION RESPONSE. This function may evolve to gather data on underlying devices.

Parameters

- **ip** – IPv4 address of KNX device.
- **port** – KNX port, default is 3671.

Returns A KNXDevice object.

Raises

- ***BOFProgrammingError*** – if IP is invalid.
- ***BOFNetworkError*** – if device cannot be reached.

bof.layers.knx.knx_functions.**group_write**(*ip: str*, *knx_group_addr: str*, *value*, *port: int = 3671*) → None

Writes value to KNX group address via the server at address ip. We first need to establish a tunneling connection so that we can reach underlying device groups.

Parameters

- **ip** – IPv4 address of KNX device.
- **knx_group_addr** – KNX group address targeted (with format X/Y/Z) Group addresses are defined in KNX project settings.
- **value** – Value to set the group address' content to.
- **port** – KNX port, default is 3671.

Returns Nothing

Raises

- ***BOFProgrammingError*** – if IP is invalid.
- ***BOFNetworkError*** – if device cannot be reached.

bof.layers.knx.knx_functions.**individual_address_scan**(*ip: str*, *addresses: object*, *port: str = 3671*) → bool

Scans KNX gateway to find if individual address exists. We first need to establish a tunneling connection and use cemi connect messages on each address to find out which one responds. As the gateway will answer positively for each address (L_data.con), we also wait for L_data.ind which seems to indicate existing addresses.

Parameters

- **ip** – IPv4 address of KNX device.
- **address** – KNx individual addresses as a string or a list.
- **port** – KNX port, default is 3671.

Returns A list of existing individual addresses.

Raises ***BOFProgrammingError*** – if IP is invalid.

Does not work (yet) for KNX gateways' individual addresses. Not reliable: Crashes after 60 addresses... Plz send help ;_; Also requires heavy refactoring after fixing issues.

bof.layers.knx.knx_functions.**line_scan**(*ip: str*, *line: str = ''*, *port: int = 3671*) → list

Scans KNX gateway to find existing individual addresses on a line. We first need to establish a tunneling connection and use cemi connect messages on each address to find out which one responds. As the gateway will answer positively for each address (L_data.con), we also wait for L_data.ind which seems to indicate existing addresses.

Parameters

- **ip** – IPv4 address of KNX device.
- **line** – KNX backbone to scan (default == empty == scan all lines from 0.0.0 to 15.15.255)
- **port** – KNX port, default is 3671.

Returns A list of existing individual addresses on the KNX bus.

Methods require smart detection of line, so far only line 1.1.X is supported and it is dirty.

bof.layers.knx.knx_functions.**search**(*ip: object = '224.0.23.12'*, *port: int = 3671*) → list

Search for KNX devices on an network using multicast. Sends a SEARCH REQUEST and expects one SEARCH RESPONSE per device.

Parameters

- **ip** – Multicast IPv4 address. Default value is default KNXnet/IP multicast address 224.0.23.12.
- **port** – KNX port, default is 3671.

Returns The list of responding KNXnet/IP devices in the network as KNXDevice objects.

Raises ***BOFProgrammingError*** – if IP is invalid.

12.2.6 Profinet DCP constants

Protocol-dependent constants (network and functions) for PNDCP.

12.3 LLDP

12.3.1 LLDP

LLDP (Link Layer Discovery Protocol) is, as its name suggests, used for network discovery directly on the Ethernet link.

BOF uses it for network discovery purposes in higher-level purposes. The implementation is imcomplete, as we only use it as a support protocol (no extended research or fuzzing intended).

Contents:

lldp_functions LLDP listen, send, create and device representation.

lldp_constants Protocol-related constants.

Uses Scapy's LLDP contrib by Thomas Tannhaeuser (hecke@naberius.de).

12.3.2 LLDP functions

Higher-level functions for network discovery using LLDP.

Contents:

LLDPDevice Object representation of a device discovered via LLDP.

Listen Sync and async functions to listen on the network for LLDP multicast requests.

Send Create basic LLDP requests and send them via multicast.

Uses Scapy's LLDP contrib by Thomas Tannhaeuser (hecke@naberius.de).

class bof.layers.lldp.lldp_functions.**LLDPDevice**(*pkt: scapy.packet.Packet = None*)

Bases: *bof.device.BOFDevice*

Object representation of a device described LLDP requests.

capabilities = None

chassis_id = None

description = None

ip_address = None

mac_address = None

name = None

parse(*pkt: scapy.packet.Packet = None*) → None
Parse LLDP response to store device information.

Parameters **pkt** – LLDP packet (Scapy), including Ethernet (Ether) layer.

port_desc = None

port_id = None

protocol = 'LLDP'

bof.layers.lldp.lldp_functions.**create_packet**(*lldp_param: dict = {'chassis_id': 'BOF', 'management_address': '0.0.0.0', 'port_desc': 'BOF discovery', 'port_id': 'port-BOF', 'system_desc': 'BOF discovery', 'system_name': 'BOF', 'ttl': 20}*) → scapy.packet.Packet
Create a LLDP packet for discovery to be sent on Ethernet layer.

Parameters **lldp_param** – Dictionary containing LLDP info to set. Optional.

bof.layers.lldp.lldp_functions.**listen_sync**(*iface: str = 'eth0', timeout: int = 20*) → list
Search for devices on an network by listening to LLDP requests.

Converts back asynchronous to synchronous with sleep (silly I know). If you want to keep asynchrone, call directly `start_listening` and `stop_listening` in your code.

bof.layers.lldp.lldp_functions.**send_multicast**(*pkt: scapy.packet.Packet = None, iface: str = 'eth0', mac_addr: str = '01:80:c2:00:00:0e'*) → scapy.packet.Packet
Send a LLDP (Link Layer Discovery Protocol) packet on Ethernet layer.

Multicast is used by default. Requires super-user privileges to send on Ethernet link.

Parameters

- **pkt** – LLDP Scapy packet. If not specified, creates a default one.
- **iface** – Network interface to use to send the packet.
- **mac_addr** – MAC address to send the LLDP packet to (default: multicast)

Returns The packet that was sent, mostly for debug and testing purposes.

bof.layers.lldp.lldp_functions.**start_listening**(*iface: str = 'eth0', timeout: int = 20*) → scapy.sendrecv.AsyncSniffer
Listen for LLDP requests sent on the network, usually via multicast.

We don't need to send a request for the others to replies, however we need to wait for devices to talk, so timeout should be high (at least 10s). Requires super-user privileges to receive on Ethernet link.

Parameters

- **iface** – Network interface to use to send the packet.
- **timeout** – Sniffing time. We have to wait for LLPD spontaneous multcast.

bof.layers.lldp.lldp_functions.**stop_listening**(*sniffer: scapy.sendrecv.AsyncSniffer*) → list

12.3.3 LLDP constants

Protocol-dependent constants (network and functions) for LLDP.

12.4 Profinet DCP

12.4.1 Profinet DCP

Profinet DCP (Discovery and COnfiguration Protocol) can be, as its name suggests, used for network discovery directly on the Ethernet link.

BOF uses it for network discovery purposes in higher-level purposes. The implementation is imcomplete, as we only use it as a support protocol (no extended research or fuzzing intended so far).

Contents:

profinet_functions Send and receive Profinet DCP identify requests and device representation.

profinet_constants Protocol-related constants.

Uses Scapy's Profinet IO contrib by Gauthier Sebaux and Profinet DCP contrib by Stefan Mehner (stefan.mehner@b-tu.de).

12.4.2 Profinet DCP functions

Higher-level functions for network discovery using PNDCP.

Contents:

PNDCPDevice Object representation of a device discovered via PNDCP.

Identify requests Send and receive identify requests and response to discover devices.

Uses Scapy's Profinet IO contrib by Gauthier Sebaux and Profinet DCP contrib by Stefan Mehner (stefan.mehner@b-tu.de).

class bof.layers.profinet.profinet_functions.**ProfinetDevice**(*pkt: scapy.packet.Packet = None*)

Bases: *bof.device.BOFDevice*

Object representation of a device responding to PN-DCP requests.

description = None

device_id = None

ip_address = None

ip_gateway = None

ip_netmask = None

mac_address = None

name = None

parse(*pkt: scapy.packet.Packet = None*) → None

protocol = 'ProfinetDCP'

vendor_id = None

bof.layers.profinet.profinet_functions.**create_identify_packet**() → scapy.packet.Packet

Create a Profinet DCP packet for discovery to be sent on Ethernet layer.

bof.layers.profinet.profinet_functions.**send_identify_request**(*iface: str = 'eth0'*, *mac_addr: str = '01:0e:cf:00:00:00'*, *timeout: int = 10*) → list

Send PN-DCP (Profinet Discovery/Config Proto) packets on Ethernet layer.

Some industrial devices such as PLCs respond to them. Multicast is used by default. Requires super-user privileges to send on Ethernet link.

Parameters

- **iface** – Network interface to use to send the packet.
- **mac_addr** – MAC address to send the PN-DCP packet to (default: multicast)
- **timeout** – Timeout for responses. More than 10s because some devices take time to respond.

12.4.3 Profinet DCP constants

Protocol-dependent constants (network and functions) for Profinet DCP.

Python Module Index

b

Index

T

U

V